arXiv:1208.4511v1 [cs.DS] 22 Aug 2012

Optimal Planar Range Skyline Reporting with Linear Space in External Memory

Yufei Tao
CUHK
Hong Kong
taoyf@cse.cuhk.edu.hk

Jeonghun Yoon
KAIST
Republic of Korea
jeonghun.yoon@kaist.ac.kr

Abstract

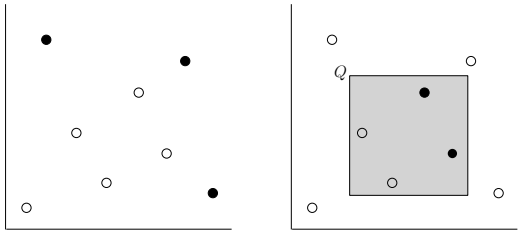
Let P be a set of n points in $\mathbb{R}^2$. Given a rectangle $Q = [\alpha_1, \alpha_2] \times [\beta_1, \beta_2]$, a *range skyline* query returns the maxima of the points in $P \cap Q$. An important variant is the so-called *top-open* queries, where Q is a 3-sided rectangle whose upper edge is grounded at $y = \infty$ (that is, $\beta_2 = \infty$). These queries are crucial in numerous database applications. In internal memory, extensive research has been devoted to designing data structures that can answer such queries efficiently. In contrast, currently there is no clear understanding about their exact complexities in external memory.

This paper presents several structures of linear size for answering the above queries with the optimal I/O cost. We show that a top-open query can be solved in $O(\log_B n + k/B)$ I/Os, where B is the block size and k is the number of points in the query result. The query cost can be made $O(\log \log_B U + k/B)$ when the data points lie in a $U \times U$ grid for some integer $U \geq n$, and further lowered to $O(1 + k/B)$ if $U = O(n)$. The same efficiency also applies to 3-sided queries where Q is a *right-open* rectangle. However, the hardness of the problem increases if Q is a *left-* or *bottom-open* 3-sided rectangle. We prove that any linear-size structure must perform $\Omega((n/B)^\epsilon + k/B)$ I/Os to solve such a query in the worst case, where $\epsilon > 0$ can be an arbitrarily small constant. In fact, left- and right-open queries are just as difficult as general (4-sided) queries, for which we give a linear-size structure with query time $O((n/B)^\epsilon + k/B)$. Interestingly, this indicates that 4-sided range skyline queries have exactly the same hardness as 4-sided range reporting (where the goal is to report simply the whole $P \cap Q$). That is, the skyline requirement does not alter the problem difficulty at all.

1 Introduction

Let p and q be two different points in $\mathbb{R}^2$, where $\mathbb{R}$ denotes the real domain. We say that p *dominates* q if $x_p \geq x_q$ and $y_p \geq y_q$, where x_p and y_p denote the x- and y-coordinates of p, respectively (similarly for x_q and y_q). Given a set P of points in $\mathbb{R}^2$, a point $p \in P$ is a *maximum* if it is not dominated by any point in P. The *skyline* of P contains all and only the maxima of P. Figure 1a shows an example where P consists of all the points, and its skyline is the set of three black points.

Given an axis-parallel rectangle Q, a *range skyline query* (also known as *range maxima query*) reports the skyline of $P \cap Q$, that is, the skyline of only the points of P covered by Q. In Figure 1b, for instance, Q is the shaded rectangle, and the two black points constitute the query result. Depending on the shape of Q, range skyline queries have several variations. Specifically, when Q is a three-sided rectangle (i.e., an edge of Q is grounded on a boundary of the universe), a range skyline query becomes a *top-open*, *right-open*, *bottom-open* or *left-open* query, as depicted in Figures 2a-2d, respectively. When Q is a two-sided rectangle whose top-right (bottom-left) corner coincides with that of the universe, it is a *dominance* (*anti-dominance*) query, shown in Figure 2e (2f). Another well-studied variation is the *contour* query, where Q is a one-sided rectangle that is the half-plane to the left of a vertical line; see Figure 2g.



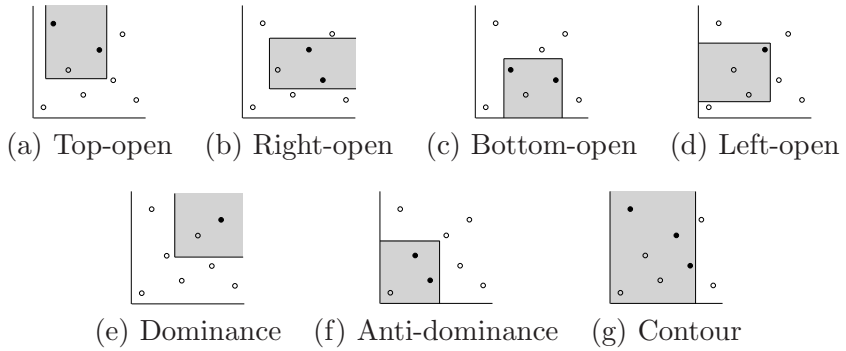
(a) Skyline (b) Range skyline

Figure 1: Range skyline queries

It is easy to observe some connections among these variations. First, top- and right-open queries are equivalent due to symmetry, and so are bottom- and left-open queries. However, top- (right-) open queries are *not* identical to bottom- (left-) queries, as we will prove later in this paper. In other words, the four types of three-sided queries are divided into two groups with distinct characteristics, which intuitively is because the skyline definition is not symmetric by all corners of the universe. Dominance, anti-dominance, and contour queries are special cases of at least one type of three-sided queries.

The focus of this paper is how to preprocess the input P into a data structure, so that range skyline queries can be efficiently answered. This has been extensively studied in theoretical computational geometry [7, 11, 12, 13, 19, 20, 22, 23, 27]. In the database area, skylines have drawn very significant attention (see [3, 5, 10, 24, 26, 29, 31, 32, 33] and the references therein) due to their crucial importance to multi-criteria optimization, which in turn is vital to a large number of applications. In particular, the rectangle of a range skyline query represents range predicates specified by a user. An effective index is essential for maximizing the efficiency of these queries in database systems [24, 29].

Unless otherwise stated, we assume that the data universe is $\mathbb{R}^2$. In practice, each dimension often has a finite domain. Formally, given an integer $U > 0$, let $[U]$ represent the set $\{0, 1, ..., U-1\}$. All the above query variations remain well defined in the universe $[U]^2$ (i.e., a $U \times U$ grid). Set



(a) Top-open (b) Right-open (c) Bottom-open (d) Left-open

(e) Dominance (f) Anti-dominance (g) Contour

Figure 2: Variations of range skyline queries (black points represent the query results)

$n = |P|$. An important degenerated case is $U = O(n)$, where the universe $[O(n)]^2$ is called the *rank space*. In general, for a smaller universe, it may be possible to achieve better query time under the same space budget. Finally, we consider that P is in general position, i.e., no two points in P have the same x- or y-coordinate. Input sets violating this condition can be easily dealt with by standard tie breaking methods. As a convention, when omitted, the base of a logarithm is assumed to be 2, i.e., $\log x = \log_2 x$.

1.1 Computation Model

Traditionally, the cost of an algorithm is measured as the amount of CPU time elapsed, as is natural when the input set can be accommodated in memory. Today, information is being accumulated at an unprecedented rate, which exceeds by far how fast the memory size of a commodity machine grows. Consequently, in many applications, the dataset cannot be contained in memory, but instead must be stored in the disk. Since calculation can happen only on data residing in (main) memory, an algorithm must perform many I/Os to move data from the disk into memory. As the speed ratio of memory to disk continuously escalates, the I/O time incurred by an algorithm dwarfs the CPU overhead to such an extent that the algorithm's cost depends almost entirely on how many I/Os are performed.

Motivated by this, we investigate range skyline queries in the *external memory* (EM) model, which was proposed in [1] and has become the dominant computation model for studying I/O-efficient algorithms. In this model, a machine has M words of memory, and a disk of an unbounded size. The disk is formatted into disjoint *blocks*, each of which is formed by B consecutive words. An I/O loads a block of data from the disk to memory, or conversely, writes B words in memory to a disk block. The space of a structure equals the number of blocks it occupies, while the time of an algorithm equals the number of I/Os it performs. CPU time is for free.

The value of M is no less than $2B$, i.e., the memory can hold at least two blocks of data. If the input set requires $O(n)$ words to store, $O(n/B)$ is referred to as the *linear* cost because it takes this many I/Os to scan the input once. Moreover, *logarithmic* cost means $O(\log_B n)$, i.e., the base should be B instead of a constant. Finally, if the universe is $[U]^2$ where U is a finite integer, a machine word is assumed to have $\Omega(\log U)$ bits. This also means that a block has $\Omega(B \log U)$ bits.

1.2 Previous Results

Range skyline in internal memory. We first review the existing results when the whole input set P fits in memory. Early research on this topic focused on dominance and contour queries,

both of which can be solved in $O(\log n + k)$ time using a structure of $O(n)$ size, where k is the number of points reported [11, 13, 19, 22, 27]. Brodal and Venkatesh [7] were the first to discover an optimal dynamic structure for top-open queries, which capture both dominance and contour queries as special cases. Their structure occupies $O(n)$ space, answers a query in $O(\log n + k)$ time, and supports an insertion and deletion in $O(\log n)$ time. The above structures belong to the *pointer machine* model. Utilizing features of the RAM model, Brodal and Venkatesh [7] also presented an alternative structure in universe $[U]^2$, which also uses $O(n)$ space, but answers a query in $O(\frac{\log n}{\log\log n} + k)$ time, and can be updated in $O(\frac{\log n}{\log\log n})$ time per insertion and deletion. It is worth mentioning that the static version of the problem can be easily settled in RAM using an RMQ (*range minimum queries*) structure (see, e.g., [39]), which uses $O(n)$ space and solves a top-open query in $O(1 + k)$ time.

For general range skyline queries (with 4-sided rectangles), all the known structures demand super-linear space. Specifically, Brodal and Venkatesh [7] gave a pointer-machine structure of $O(n \log n)$ size, $O(\log^2 n+k)$ query time, and $O(\log^2 n)$ update time. Kalavagattu et al. [20] designed a static RAM-structure that occupies $O(n \log n)$ space and achieves query time $O(\log n + k)$. In the rank space $[O(n)]^2$, Das et al. [12] proposed a static RAM-structure with $O(n\frac{\log n}{\log\log n})$ space and $O(\frac{\log n}{\log\log n} + k)$ query time.

Although the above results also hold directly in external memory, they are far from being satisfactory. In particular, all of them incur $\Omega(k)$ I/Os to report a query result of k points. An I/O-efficient structure ought to achieve $O(k/B)$ I/Os for the same purpose.

Range skyline in external memory. In contrast to internal memory where there exist a large number of results, range skyline queries have not been well studied in external memory. As a naive solution, we can first scan the entire P to eliminate the points falling outside the query rectangle Q, and then find the skyline of the remaining points by the fastest skyline algorithm [33] on non-preprocessed input sets. This solution is very expensive, and can incur $O((n/B)\log_{M/B}(n/B))$ I/Os. Papadias et al. [29] described a branch-and-bound algorithm when the dataset is indexed by an R-tree [4, 16]. The algorithm is heuristic in nature and cannot guarantee better query time than the naive solution mentioned earlier in the worst case.

Very recently, Kejberg-Rasmussen et al. [23] designed the first I/O-efficient structure for top-open queries. For any ϵ satisfying $0 \leq \epsilon \leq 1$, their structure occupies $O(n/B^{1-\epsilon})$ space, answers a query in $O(\log_{2B^\epsilon} n + k/B^{1-\epsilon})$ I/Os, and supports an update in $O(\log_{2B^\epsilon} n)$ I/Os. For our discussion, the most interesting tradeoff is obtained with $\epsilon = 0$, in which case the structure uses linear space, has $O(\log_2 n + k/B)$ query time and $O(\log_2 n + k/B)$ update time. For $\epsilon > 0$, the structure requires more than linear space by a large factor B^ϵ, while the query cost is higher than the desired $O(\log_B n + k/B)$ bound also by B^ϵ times for large k. No solution is known for 4-sided range skyline queries.

Pointer-machine lower bound. An astute reader would have noticed that all the above results for three-sided queries focus exclusively on top-open (equivalently, right-open) queries. In particular, no structure for left-open (equivalently, bottom-open) queries can match the space-query tradeoff of any top-open structure aforementioned. This is not accidental, at least not under the (internal memory) pointer machine, according to a lower bound result of Kejberg-Rasmussen et al. [23]. They constructed a set of hard anti-dominance queries over a low-discrepancy point set by Chazelle [8]. Combining this query set with a result of Chazelle and Liu [9], they proved that, for any structure to answer an anti-dominance query in $O(\log^c n + k)$ time for any constant $c > 0$, the space consumption must be $\Omega(n\frac{\log n}{\log\log n})$. Since anti-dominance queries are specialization of left-open queries, the same lower bound also applies to the latter, namely, no structure of $O(n)$

size can solve a left-open query in $O(\log n + k)$ time. In external memory, this implies that a pointer-machine structure of linear size cannot guarantee $O(\frac{1}{B}\log n + k/B)$ query time. No lower bound, however, is known for I/O-efficient structures outside the pointer machine.

Other related results in external memory. Range skyline queries stem from the marriage of skyline and *range queries*. Specifically, given an axis-parallel rectangle Q, a range query reports all the points of the dataset P that are covered by Q. These queries have been well understood in external memory. There exists a linear-size structure that answers a range query in $O((n/B)^\epsilon + k/B)$ I/Os [15, 21, 35]. As another interesting tradeoff, a structure of [2] uses $O(\frac{n}{B}\frac{\log n}{\log\log_B n})$ space and solves a query in $O(\log_B n + k/B)$ I/Os. Both of the above tradeoffs are optimal, according to the lower bounds of [2, 17, 21, 34, 35]. This means that one cannot hope to achieve query time $O(\log_B n + k/B)$ with a structure of linear size.

Three-sided range queries are an important variant where Q is a three-sided rectangle (same as those in Figures 2a-2d). Note that since no corner-asymmetry (such as dominance) exits for range queries, the orientation of Q is irrelevant. Namely, no matter which side of Q is open, all three-sided range queries can be supported by an identical structure after rotation. Occupying linear space, the *external priority search tree* [2] is able to answer a three-sided query in $O(\log_B n + k/B)$ I/Os.

Most research in external memory makes the *indivisibility assumption*. Informally, this assumption requires that a data structure should store every point coordinate as an atom. We are not allowed, for example, to cut the bits of a coordinate into multiple pieces, and place them into different blocks; neither can we compress the coordinate to save space. A justification of the assumption is that it holds for most practical structures (e.g., B-trees, R-trees, kd-trees, etc.). Another more theoretical justification is that it facilitates the development of lower bounds. Indeed, in external memory, all existing lower bounds of range queries [2, 17, 21, 34, 35] were derived with this assumption in mind[1].

Efforts have been made in recent years towards understanding I/O-efficient algorithms without the indivisibility assumption [18, 25, 30, 37]. The general observation is that, when the universe is finite (e.g., $[U]^2$ for some integer U), sometimes it is possible to derive a space-query tradeoff better than the optimal tradeoff in an infinite universe like $\mathbb{R}^2$. As an example related to our work, Larsen and Pagh [25] shows that, in the rank space $[O(n)]^2$, there is a linear-size structure that answers a 3-sided range query in $O(1 + k/B)$ I/Os, i.e., shaving off the additive factor term $O(\log_B n)$ in the query time of the external priority search tree.

1.3 Our Results

It is clear from the above discussion that, currently there is no clear understanding about the exact complexities of range skyline queries in external memory. This paper addresses this issue by presenting a set of linear-size structures with optimal query efficiency, as elaborated below.

First, for top-open queries, we give an elegant reduction that converts the problem to *segment intersection*. Specifically, in the latter problem, the input is a set S of horizontal segments in $\mathbb{R}^2$. Given a vertical segment q, a query reports all the segments of S intersecting q. The segment intersection problem can be settled by a *persistent B-tree* [28]. It immediately implies a linear-size structure that answers a top-open query optimally in $O(\log_B n + k/B)$ I/Os (Theorem 1). The result applies to dominance and contour queries as well, since they are specialization of top-open queries.

As a second step, we prove an interesting feature of our top-open structure: it can be constructed

[1]This is not completely true for [35], whose results hold under a weaker form of the assumption.

in $O(n/B)$ I/Os after the input set P has been sorted on x-coordinates (Theorem 1). We call this property *sort-aware build-efficient* (SABE), which in general says that a structure can be built in linear I/Os after the input elements have been arranged in a certain order. The importance of SABE is that, it singles out the most expensive step (i.e., sorting) from the rest of the construction. A non-trivial SABE example is the external priority search tree: Tao [36] showed how to utilize this property to reduce the cost of updating a structure where the external priority search tree is deployed as a secondary structure. We prove that our top-open structure is SABE by designing a linear-time algorithm for constructing a persistent B-tree. This is interesting because persistent B-trees are not SABE in general [38]. Our algorithm is assisted by several intrinsic properties of top-open queries that are established for the first time in this paper.

The above structures are *indivisible*, namely, they obey the indivisibility assumption (this fact has the significance that the aforementioned results can be achieved without using the extra power allowed by breaking the assumption). Next, we improve the query time beyond the logarithmic bound when the data universe is small. Specifically, when the universe is $[U]^2$ where U is an integer, we give a *divisible* linear-size structure with query time $O(\log\log_B U + k/B)$, which is optimal (Corollary 1). In the rank space, the query time can be further reduced to $O(1 + k/B)$ (Theorem 2). These results are based on a new divisible structure that answers a *ray dragging* query in $O(1)$ I/Os on a small point set P (Lemma 4). Specifically, given a vertical ray ρ, a ray dragging query reports the first point in P hit by moving ρ leftwards.

Our final contribution targets anti-dominance, bottom-open (hence, left-open), and the most general 4-sided queries. We prove that all of them actually have exactly the same hardness as far as indivisible linear-size structures are concerned. Specifically, any such structure must incur $\Omega((n/B)^\epsilon + k/B)$ I/Os answering a query in the worst case (Corollary 2), where $\epsilon > 0$ can be an arbitrarily small constant. Furthermore, this is tight because there is a linear-size structure matching this efficiency (Theorem 4). Recall that $\Theta((n/B)^\epsilon + k/B)$ is also the optimal query time of range queries under the linear space budget (see Section 1.2). Therefore, interestingly, range skyline queries are just as hard as range queries, i.e., the skyline requirement does not change the problem difficulty at all.

In all cases, our query algorithms report the points of a query result by the order of y-coordinates (equivalently, by the order of x-coordinates). This is not trivial because if the reporting order is not guaranteed, obtaining it requires sorting in $O(\frac{k}{B}\log_{M/B}\frac{k}{B})$ I/Os.

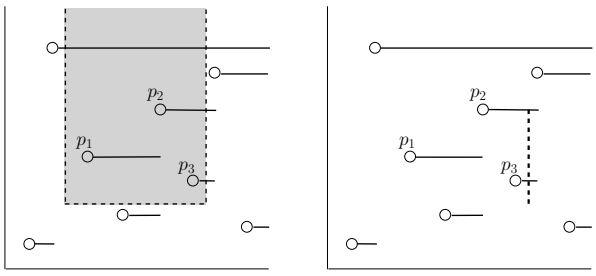
2 SABE Top-open Structure

This section will describe a structure of linear size that solves a top-open query in $\mathbb{R}^2$ using $O(\log_B n + k/B)$ I/Os. The structure is SABE, i.e., it can be built in $O(n/B)$ I/Os if the input set P has been sorted by x-coordinates.

Throughout the paper, we will make frequent use of B-trees. Unless otherwise stated, a B-tree has both *leaf capacity* and *internal fanout* set to B. Specifically, the former gives the maximum number of elements that can be stored in a leaf node, whereas the latter specifies the maximum number of child nodes that an internal node can have.

2.1 Reduction to Segment Intersection

We first describe a simple linear-size structure with $O(\log_B n + k/B)$ query time. This is achieved with an elegant reduction that converts a top-open query to a query of segment intersection.



(a) Data conversion (b) Converted query

Figure 3: Reduction

Let p be any point in P. Denote by *leftdom*(p) as the leftmost point among all the points in P dominating p. If such a point does not exist, *leftdom*(p) = NULL. We convert p to a horizontal segment $\sigma(p)$ as follows. Let q = *leftdom*(p). If q = NULL, then $\sigma(p) = [x_p, \infty) \times y_p$, that is, the horizontal segment whose x-span is interval $[x_p, \infty)$, and y-projection is y_p. Otherwise (i.e., q exists), $\sigma(p) = [x_p, x_q) \times y_p$. Define $\Sigma(P) = \{\sigma(p) \mid p \in P\}$, i.e., the set of segments converted from the points of P.

Now, consider a top-open query with rectangle $Q = [\alpha_1, \alpha_2] \times [\beta, \infty)$. We answer it by performing segment intersection on $\Sigma(P)$. First, obtain β' as the highest y-coordinate of the points in $P \cap Q$. Then, report all segments in $\Sigma(P)$ that intersect the vertical segment $\alpha_2 \times [\beta, \beta']$.

To illustrate the reduction, Figure 3a shows the segments in $\Sigma(P)$, where P is the set of points in Figure 1a. For example, p_2 = *leftdom*(p_1), which is why $\sigma(p_1)$ terminates at x_{p_2}. The shaded rectangle represents the search area Q of a top-open query. In this case, the value of β' equals y_{p_2}. Hence, our algorithm creates the dashed vertical segment shown in Figure 3b, and finds all the segments of $\Sigma(P)$ intersecting it (i.e., $\sigma(p_2)$ and $\sigma(p_3)$). It is easy to verify that the query result is $\{p_2, p_3\}$. The lemma below formally establishes the correctness of our algorithm.

Lemma 1. *Our algorithm correctly answers all top-open queries.*

Proof. Consider any point $p \in P$. We show that our algorithm reports p if and only if p satisfies the top-open query with search area $Q = [\alpha_1, \alpha_2] \times [\beta, \infty)$.

If direction: As p satisfies the query, we know that $p \in Q$, $y_p \leq \beta'$, and q = *leftdom*$(p) \notin Q$. The last fact suggests that $x_q > \alpha_2$ (in the special case q = NULL, define $x_q = \infty$). Hence, $\sigma(p) = [x_p, x_q) \times y_p$ intersects the vertical segment $\alpha_2 \times [\beta, \beta']$, and thus, will be reported by our algorithm.

Only-if direction: Let p be a point found by our algorithm, i.e., $\sigma(p) = [x_p, x_q) \times y_p$ intersects $\alpha_2 \times [\beta, \beta']$, where q = *leftdom*(p). It follows that $x_p \leq \alpha_2 < x_q$ and $\beta \leq y_p \leq \beta'$.

Next, we prove $\alpha_1 \leq x_p$. Recall that β' is the y-coordinate of the highest point p' among all the points in $P \cap Q$. If $p = p'$, then $\alpha_1 \leq x_p$ clearly holds. Otherwise, we know $y_p < y_{p'}$, which implies that $x_p > x_{p'}$. This is because if $x_p < x_{p'}$, then p' dominates p, which (because $x_{p'} \leq \alpha_2 < x_q$) violates the definition of q. Now, $x_p \geq \alpha_1$ follows from $x_{p'} \geq \alpha_1$.

So far we have shown that p is covered by Q. It remains to prove that p is not dominated by any point in $P \cap Q$. This is true because $\alpha_2 < x_q$ suggests that the leftmost point in P dominating p must be outside Q. □

Note that β' can be easily found in $O(\log_B n)$ I/Os with a *range-max query* on a slightly augmented B-tree indexing the x-coordinates in P. To facilitate retrieving the segments intersecting

$\alpha_2 \times [\beta, \beta']$, we store $\Sigma(P)$ in a persistent B-tree [28]. As $\Sigma(P)$ has n segments, the persistent B-tree occupies $O(n/B)$ space and answers a segment intersection query in $O(\log_B n + k/B)$ I/Os. We thus have obtained a linear-size top-open structure with $O(\log_B n + k/B)$ query time.

More efforts, however, are needed to make the structure SABE. In particular, we have to overcome two challenges. First, we must generate $\Sigma(P)$ in linear I/Os. Second, the persistent B-tree on $\Sigma(P)$ must be built in the same amount of time. We explain how to achieve these purposes in the subsequent sections. It is worth noting that the (augmented) B-tree for the computation of β' can be easily built in linear I/Os after P is sorted by x-coordinates.

2.2 Computing $\Sigma(P)$

$\Sigma(P)$ cannot be an arbitrary set of segments. Next, we reveal two properties about it, which are behind the correctness and efficiency of our algorithms.

Lemma 2. *$\Sigma(P)$ has the following properties:*

- **(Nesting)** *for any two segments s_1 and s_2 in $\Sigma(P)$, their x-intervals are either disjoint, or such that one x-interval contains the other.*
- **(Monotonic)** *let ℓ be any vertical line, and $S(\ell)$ the set of segments in $\Sigma(P)$ intersected by ℓ. If we order the segments in $S(\ell)$ bottom-up by y-coordinates, the lengths of their x-intervals increase monotonically.*

Proof. Nesting: Let p_1 and p_2 be the points such that $s_1 = \sigma(p_1)$ and $s_2 = \sigma(p_2)$. Assume without loss of generality that $x_{p_1} < x_{p_2}$. Consider first the case $y_{p_1} < y_{p_2}$. In this scenario, the x-interval of s_1 must terminate before x_{p_2} because p_2 dominates p_1. In other words, s_1 and s_2 have disjoint x-intervals.

We now discuss the case $y_{p_1} > y_{p_2}$. If $\mathit{leftdom}(p_1)$ has x-coordinate smaller than x_{p_2}, s_1 and s_2 have disjoint x-intervals. Otherwise, $\mathit{leftdom}(p_1)$ also dominates p_2, implying that the x-interval of s_2 is enclosed in that of s_1.

Monotonic: Let ℓ intersect the x-axis at α. Consider the contour query with rectangle $Q = (-\infty, \alpha] \times (-\infty, \infty)$, which is a special top-open query. By Lemma 1, the left endpoints of the segments in $S(\ell)$ constitute the skyline of $P \cap Q$. Therefore, if we enumerate the segments of $S(\ell)$ in ascending order of y-coordinates, their left endpoints' x-coordinates decrease continuously. It thus follows from the nesting property that their x-intervals have increasing lengths. □

We are ready to present our algorithm for computing $\Sigma(P)$, after P has been sorted by x-coordinates. Conceptually, we sweep a vertical line ℓ from $x = -\infty$ to ∞. At any time, the algorithm (essentially) stores the set $S(\ell)$ of segments in a stack (recall the definition of $S(\ell)$ in Lemma 2). More specifically, the segments of $S(\ell)$ are en-stacked in descending order of y-coordinates (i.e., the highest segment enters the stack first). Whenever a segment is popped out of the stack, its right endpoint is decided, and output. In general, the segments of $\Sigma(P)$ are output in ascending order of their right endpoints' x-coordinates.

We now give the algorithm's details. It starts by pushing the first (i.e., leftmost) point of P into the stack. Iteratively, let p be the next point fetched from P. We check whether $y_p > y_q$, where q is the point currently at the top of the stack. If yes, we know that $p = \mathit{leftdom}(q)$. Hence, the algorithm pops q out of the stack, and outputs segment $\sigma(q) = [x_q, x_p) \times y_q$. Then, setting q to the point that tops the stack now, the algorithm checks again whether $y_p > y_q$, and repeats the above

steps if yes. This continues until either the stack is empty or $y_p < y_q$. In either case, the iteration finishes by pushing q into the stack.

It is clear from the earlier discussion that the algorithm generates $\Sigma(P)$ in $O(n/B)$ I/Os.

2.3 Constructing the Persistent B-tree

Remember that we need a persistent B-tree T on $\Sigma(P)$. Usually, the construction of a persistent B-tree requires super-linear I/Os even after sorting [38]. Below, we show that the two properties of $\Sigma(P)$ in Lemma 2 allow building T in linear I/Os.

Let us number the leaf level as *level 0*. In general, the parent of a level-i ($i \geq 0$) node is at level $i + 1$. We will build T in a bottom-up manner, i.e., starting from the leaf level, then leave 1, and so on. We will first explain how to create the leaf nodes from $\Sigma(P)$.

We will again apply plane sweep, for which purpose we need to sort the left and right endpoints (of the segments) in $\Sigma(P)$ together by their x-coordinates. This can be done in $O(n/B)$ I/Os as follows. First, P, which is sorted by x-coordinates, essentially gives a sorted list of the left endpoints in $\Sigma(P)$. On the other hand, our algorithm of the previous subsection generates $\Sigma(P)$ in ascending order of the right endpoints. By merging the two lists in linear time, we obtain the desired sorted list of left and right endpoints combined.

Before elaborating our approach of building the persistent B-tree, let us briefly review the traditional algorithm proposed in [28]. The algorithm conceptually moves a vertical line ℓ from $x = -\infty$ to ∞. At any moment, it maintains a B-tree $T(\ell)$ on the y-coordinates of the segments in $S(\ell)$ (recall that $S(\ell)$ is the set of segments in $\Sigma(P)$ intersecting ℓ). We call $T(\ell)$ the *snapshot B-tree*. To do so, whenever ℓ hits the left (right) endpoint of a segment s, it inserts (deletes) the y-coordinate of s in $T(\ell)$. Deletions are logical, i.e., they simply mark the positions of ℓ at which the corresponding elements are deleted, instead of physically discarding those elements. Overall, the persistent B-tree can be regarded as a space-efficient union of all the snapshot B-trees.

The above algorithm incurs $O(n \log_B n)$ I/Os because, intuitively, (i) there are $2n$ updates in total, and (ii) for each update, $O(\log_B n)$ I/Os are needed to locate the leaf node to be modified. It turns out that when $\Sigma(P)$ is nesting and monotonic, the construction can be significantly accelerated. The most crucial observation is that any update to $S(\ell)$ happens only *at the bottom* of ℓ. Specifically, whenever ℓ hits the left/right endpoint of a segment $s \in \Sigma(P)$, s must be the lowest segment in $S(\ell)$ (otherwise, either the nesting or monotonicity property is violated). This implies that the leaf node of $T(\ell)$ to be altered must be the first one in $T(\ell)$ (as it contains s). Hence, we can find this leaf without any I/O by buffering it in memory, in contrast to the $O(\log_B n)$ cost originally needed.

The other details are standard. We sketch them below assuming that the reader is familiar with the algorithm of [28]. First, since we focus on only the leaf level, it is unnecessary to maintain any internal nodes. Whenever the first leaf u of $T(\ell)$ is full, we version copy it to u', and possibly perform a split on u', or merge u' with its sibling (in this case, the sibling needs to be version copied first). The sibling can be found in one I/O by keeping a pointer to it in u. In general, such sibling pointers are created in node splits, and properly maintained during version copies and merges. By the standard analysis [28], a version copy, split, and merge can all be handled in $O(1)$ I/Os, and can happen only $O(n/B)$ times. Therefore, the cost of building the leaf level is $O(n/B)$.

Now we explain how to build the nodes of level 1. This can in fact be achieved by exactly the same algorithm as described above, but on a different set of segments. To explain, we first review an intuitive way [28] to visualize a node u in a persistent B-tree. The node can be viewed as a

rectangle $r(u)$ in $\mathbb{R}^2$. Specifically, the x-interval of $r(u)$ starts (ends) at the position of ℓ at which u is created (version copied). The y-interval of $r(u)$ starts (resp., ends) at the smallest y-coordinate in u (resp., in the succeeding sibling of u) in the snapshot B-tree where u is created.

Let Σ_1 be the set of bottom edges of all $r(u)$, where u ranges over all the leaf nodes of the persistent B-tree already obtained. Note, however, that if a bottom edge is $[x, x'] \times y$, we include $[x, x') \times y$ into Σ_1, namely, the right endpoint of the edge is not taken. $|\Sigma_1| = O(n/B)$, i.e., the number of leaf nodes. The next lemma points out a crucial fact. To enhance presentation, we have moved some relatively standard proofs (such as the one of the lemma below) to the appendix.

Lemma 3. *Σ_1 is both nesting and monotonic.*

Notice that our algorithm (for building the leaf nodes) writes the left and right endpoints of the segments in Σ_1 in ascending order of their x-coordinates, as is due to the plane sweep. This, together with Lemma 3, permits us to create the level-1 nodes using the same algorithm in $O(n/B^2)$ I/Os (recall that $|\Sigma_1| = O(n/B)$). We repeat the above process to construct the nodes of higher levels. The cost decreases by a factor of B every level up. The overall construction time is therefore $O(n/B)$. We are now ready to prove our first main result:

Theorem 1. *There is an indivisible linear-size structure on n points in $\mathbb{R}^2$ such that, a top-open range skyline query can be answered in $O(\log_B n + k/B)$ I/Os, where k is the number of reported points. The points of the query result are reported in the order of y-coordinates. If all points have been sorted by x-coordinates, the structure can be built in linear I/Os. The query time is optimal even without the indivisibility assumption.*

Proof. We first prove the part about report ordering. As discussed in Section 2.1, we answer a top-open query by retrieving the segments of $\Sigma(P)$ intersecting a vertical segment. Using the persistent B-tree, we can do so by listing those segments in ascending order of their y-coordinates [28]. This establishes the desired output ordering because the left endpoints of those segments constitute the result of the top-open query.

Next, we discuss the query time's optimality. First, the k/B term is indispensable if k points need to be reported. The $O(\log_B n)$ term, on the other hand, is also compulsory as can be shown by a reduction from predecessor search. Precisely speaking, the reduction is in fact from predecessor search to top-open range queries (note: *not* range skyline queries), which is well known in the literature (see, e.g., [6]). Specifically, if a linear-size structure can answer a top-open range query in $f(n, B) + O(k/B)$ time, the same structure also solves a predecessor query in $f(n, B)$ time. Interestingly, given a predecessor query, the converted top-open range query always returns only 1 point. Hence, the query can also be interpreted as a top-open range skyline query, i.e., the same reduction also works from predecessor search to top-open range skyline queries. Finally, any linear-size structure must incur $\Omega(\log_B n)$ I/Os answering a predecessor query in the worst case [30]. It thus follows that $\Omega(\log_B n)$ also lower bounds the cost of a top-open query.

The rest of the theorem follows from the earlier discussion directly. □

3 Divisible Top-open Structure

The structure of the preceding section does not divide any coordinate, i.e., Theorem 1 holds even under the *computationally-weaker* external memory model with the indivisibility assumption. This section eliminates the assumption, and unleashes the power endowed by bit manipulation. As we will see, when the universe is small, it leads to linear-size structures with faster query time than in Theorem 1.

3.1 Ray Dragging

We now take a short break from range skyline queries to discuss the *ray dragging* problem. The input is a set S of m points in $[U]^2$ where $U \geq m$ is an integer. Given a vertical ray $\rho = \alpha \times [\beta, U]$ where $\alpha, \beta \in [U]$, a ray dragging query reports the first point in S that is hit by ρ when ρ moves towards left. We want to store S in a structure to answer all queries efficiently. We will prove:

Lemma 4. *For $m = (B \log U)^{O(1)}$, we can store S in a structure of size $O(m/B)$ such that a ray dragging query can be answered in $O(1)$ I/Os.*

Recall that a machine word has $\Omega(\log U)$ bits, and hence a block has $\Omega(B \log U)$ bits. The remainder of this subsection serves as the proof of Lemma 4. We will adopt the framework of [25] that was used to design a structure for 3-sided range queries. Nonetheless, as will be clear shortly, several ideas specific to ray dragging are required to obtain our result.

A structure with query time $O(\log_B m)$. For this purpose, we simply store S in a B-tree that indexes the x-coordinates of the points in S. Let u be an internal node whose child nodes are $v_1, ..., v_B$. For each $i \in [1, B]$, we store with u a point $Y_{max}(v_i)$, where in general $Y_{max}(v)$ gives the highest point in the subtree of v. Define $Y^{\star}_{max}(u) = \{Y_{max}(v_i) \mid 1 \leq i \leq B\}$. Furthermore, for a leaf node z, define $Y^{\star}_{max}(z)$ to be the set of points stored in z.

We answer a ray-dragging query with ray $\rho = \alpha \times [\beta, U]$ as follows. First, descend a root-to-leaf path π to the leaf node containing the predecessor of α among the x-coordinates (of the points) in S. Let u be the *lowest* node on π such that $Y^{\star}_{max}(u)$ has a point that can be hit by ρ when ρ moves left. Note that whether $Y^{\star}_{max}(u)$ includes such a point can be checked in $O(1)$ I/Os by loading $Y^{\star}_{max}(u)$ into memory. Hence, u can be identified in $O(h)$ I/Os where h is the height of the B-tree. If u does not exist, we return an empty result (i.e., ρ does not hit any point no matter how far it moves).

For the case where u exists, let p be the first point in $Y^{\star}_{max}(u)$ hit by the left-moving ρ. Suppose that p is in the subtree of v, where v is a child node of u. The query result must be in the subtree of v, although it may not necessarily be p. To find out, we descend another path from v to a leaf. Specifically, we reset u to v, and find the first point p in $Y^{\star}_{max}(u)$ $(= Y^{\star}_{max}(v))$ that is hit by the left-moving ρ (notice that p has changed). Now, letting v be the child node of u from whose subtree p is from, we repeat the above steps. This continues until u becomes a leaf, in which case the algorithm returns p as the final answer.

It is easy to see that the query time is $O(h) = O(\log_B m)$. We will refer to the above structure as a *ray-drag B-tree*. Note that if $B \geq \sqrt{\log U}$, $O(\log_B m) = O(\log_B(B \log U)) = O(1)$, which fulfills the purpose of Lemma 4. We therefore focus on $B < \sqrt{\log U}$ in the sequel.

Minute structure. Set $b = B \log U$. As $B < \sqrt{\log U}$, $b < \log^{3/2} U$. We now consider the case where S has very few points, or specifically, $m \leq \sqrt{b} < \log^{3/4} U$.

We convert S into a set S' of points in a $[m]^2$ grid. For this purpose, we map a point $p \in S$ to $p' \in S'$ such that p'_x (p'_y) is the number of points in S whose x- (y-) coordinates are at most p_x (p_y). That is, p'_x (p'_y) is the *rank* of p_x (p_y) among the x- (y-) coordinates in S.

Given a ray $\rho = \alpha \times [\beta, \infty)$, we instead answer a query in $[m]^2$ using a ray $\rho' = \alpha' \times [\beta', \infty)$ obtained from ρ. Specifically, α' (β') is the rank of the predecessor of α (β) among the x- (y-) coordinates in S. We create a *fusion tree* [14] on the x- (y-) coordinates in S so that the predecessor of α (β) can be found in $O(\log_b m) = O(1)$ I/Os (see also [25]), which is thus also the cost of turning ρ into ρ'. The fusion tree uses $O(m/B)$ blocks.

We will ensure that the query with ρ' (in $[m]^2$) returns an id from 1 to m that uniquely identifies a point p in S, if the result is non-empty. To convert the id into the coordinates of p, we store the points of S in an array such that any point can be retrieved in one I/O by its id. Clearly, the array occupies $O(m/B)$ blocks.

Next, we explain how to store S'. The benefit of working with S' is that each coordinate in $[m]^2$ requires fewer bits to represent than one in $[U]^2$, i.e., $\log_2 m$ bits as opposed to $\log_2 U$. In particular, we need $3\log_2 m$ bits in total to represent a point's x-, y-coordinates, and id. Since $|S'| = m$, the storage of the entire S' demands $3m \log m = O(\log^{3/4} U \cdot \log\log U) = o(\log U)$ bits. In other words, we can store the entire S' in a single word! Given a query with ρ', we simply load this word into memory, and answer the query in memory with no more I/O.

From the above description, we have obtained a structure of $O(m/B)$ blocks with constant query time when $m \leq \sqrt{b}$.

Proving Lemma 4. We are ready to discuss the case where $m = b^{O(1)}$, as needed in Lemma 4. In this case, we create a ray-drag B-tree on S as described earlier, but setting its internal fanout to b (the leaf capacity is still B). The height of the tree is therefore $h = O(\log_b m) = O(1)$. Recall that each internal node u of the ray-drag B-tree is associated with a set $Y^\star_{max}(u)$ of size b. We can no longer store $Y^\star_{max}(u)$ in a single block because b may be greater than B. Instead, we create a minute structure on $Y^\star_{max}(u)$, which needs $O(1+b/B)$ space. Since there are $O(\min\{n/B, n/(bB)\})$ internal nodes, all the minute structures occupy $O(n/B)$ blocks in total.

A query with ray ρ can still be answered by the same algorithm of the ray-drag B-tree discussed before. The only difference is that, at an internal node u, we search the minute structure on $Y^\star_{max}(u)$ to find the first point p hit by the left-moving ρ. As this requires only $O(1)$ I/Os, the total query cost is $O(h) = O(1)$. There is a technical detail that requires a bit of clarification. Recall that, regarding p, we need to know which child node of u contains p in the subtree. This can be achieved by associating p with the child node's index in u. The index requires only $\log_2 b \leq \log_2 U$ bits, which is no more than the length of a coordinate. Hence, we can easily store the index along with p in the minute structure on $Y^\star_{max}(u)$. We thus complete the proof of Lemma 4.

3.2 Top-open Structure on Few Points

We now resume our study of top-open queries. Remember that the input is a set P of n points in $[U]^2$ for some integer $U \geq n$, and a query is given a rectangle $Q = [\alpha_1, \alpha_2] \times [\beta, U]$ where $\alpha_1, \alpha_2, \beta \in [U]$. We will present a structure for small P, specifically:

Lemma 5. *For $n \leq (B \log U)^{O(1)}$, we can store P in a linear-size structure that answers a top-open range skyline query in $O(1 + k/B)$ I/Os, where k is the number of reported points. Furthermore, the points are reported in the order of y-coordinates.*

Proof. Consider a query with $Q = [\alpha_1, \alpha_2] \times [\beta, U]$. Let p be the first point hit by the ray $\rho = \alpha_2 \times [\beta, U]$ when ρ moves left. If p does not exist or is out of Q (i.e., $p_x < \alpha_1$), the top-open query has an empty result. Otherwise (i.e., $p \in Q$), p must be the lowest point in the skyline of $P \cap Q$.

The subsequent discussion focuses on the scenario where $p \in Q$. We index $\Sigma(P)$ with a persistent B-tree T, as in Theorem 1. Recall that the top-open query can be solved by retrieving the set S of segments in $\Sigma(P)$ intersecting the vertical segment $\psi = \alpha_2 \times [\beta, \beta']$, where β' is the highest y-coordinate of the points in $P \cap Q$. To do so in $O(1 + k/B)$ I/Os, we need some observations:

1. *S includes exactly the segments of $\Sigma(P)$ intersecting the vertical segment $\psi' = p_x \times [p_y, \beta']$.*

We prove this in two steps. First, notice that $\sigma(p)$ is the lowest among the segments of $\Sigma(P)$ intersecting ψ (recall that $\sigma(p)$ is the segment in $\Sigma(P)$ converted from p). Hence, a segment of $\Sigma(P)$ intersects ψ if and only if it intersects $\alpha_2 \times [p_y, \beta']$. Second, a segment of $\Sigma(P)$ intersects $\alpha_2 \times [p_y, \beta']$ if and only if it intersects ψ'. This is because of the nesting and monotonicity properties of $\Sigma(P)$. Specifically, let $s \neq \sigma(p)$ be a segment in $\Sigma(P)$ intersecting $\alpha_2 \times [p_y, \beta']$. As s is higher than $\sigma(p)$, the x-interval of s must contain that of $\sigma(p)$, implying that s intersects ψ'. In the same manner, one can show that if s intersects ψ', it also intersects $\alpha_2 \times [p_y, \beta']$.

2. *Let $T(\ell)$ be the snapshot B-tree in T when ℓ is at the position $x = p_x$. Once we have obtained the leaf node in $T(\ell)$ containing y_p, we can retrieve S in $O(1 + k/B)$ I/Os without knowing the value of β'.*

   Each leaf node in $T(\ell)$ has a sibling pointer to its succeeding leaf node. Hence, starting from the leaf storing y_p, we can visit the leaves of $T(\ell)$ in ascending order of the y-coordinates they contain. The effect is to report in the bottom-up order the segments of $\Sigma(P)$ that intersect the ray $p_x \times [p_y, U]$. By nesting and monotonicity, the left endpoint of a segment reported latter has a smaller x-coordinate. We stop as soon as reaching a segment whose left endpoint falls out of Q. The cost is $O(1 + k/B)$ because $\Omega(B)$ segments are reported in each accessed leaf, except possibly the last one.

We now elaborate the structure of Lemma 5. Besides T, also create a structure of Lemma 4 on P. Moreover, for every point $p \in P$, keep a pointer to the leaf node that (i) is in the snapshot B-tree $T(\ell)$ when ℓ is at $x = p_x$, and (ii) contains y_p. Store the pointers in an array of size n to permit retrieving the pointer of any point in one I/O. The query algorithm is straightforward from the previous analysis, and performs $O(1)$ I/Os. □

We will refer to the structure of Lemma 5 as a *few-point structure*.

3.3 Final Top-Open Structure

This subsection proposes our top-open structure that works for arbitrary n. For simplicity, we will discuss first the rank space, i.e., the universe is $[U]^2$ where $U = O(n)$. Our results will be extended to general U at the end of the section.

Structure. A part of our solution externalizes a pointer-machine structure in [7]. That structure, however, has logarithmic query time. Hence, extra ideas are needed to eliminate the logarithmic factor. Below we show how to achieve this with Lemma 5.

We assume, without loss of generality, that both $\lambda = B \log^2 U$ and U/λ are integers. Divide the x-dimension of $[U]^2$ into U/λ consecutive intervals of length λ each. Call each interval a *chunk*. Assign each point $p \in P$ to the unique chunk covering x_p. Note that some chunks may be empty.

Create a complete binary search tree $\mathcal{T}$ on the chunks. Let u be a node of $\mathcal{T}$. Denote by $P(u)$ the set of points (assigned to the chunks) in the subtree of u. Define $high(u)$ as the set of B highest points in the skyline of $P(u)$. If the skyline of $P(u)$ has less than B points, $high(u)$ includes all of them. Furthermore, if $|high(u)| = B$, let $highend(u)$ be the lowest point in $high(u)$; otherwise, $highend(u) =$ NULL. We store $high(u)$ along with u. Also, if $highend(u) \neq$ NULL, we record in u a pointer the chunk covering the x-coordinate of $highend(u)$.[2]

[2]The pointer is not needed in the rank space. Later, the same structure as described here will be deployed in a universe $[U]^2$ of general U, where the pointer will be useful.

Consider an internal node u such that $p = highend(u)$ is not NULL. In this case, let $\pi(u)$ be the path from the leaf (a.k.a. chunk) z of $\mathcal{T}$ covering x_p to the child of u that is an ancestor of z. Define $\Pi_\gamma(u)$ as the set of right siblings of the nodes in $\pi(u)$ (note: if a node is the right child of its parent, it has no right sibling; similarly, if a node is a left child, it has no left sibling). Let $MAX(u)$ be the skyline of the point set

$$\bigcup_{v \in \Pi_\gamma(u)} high(v).$$

We store $MAX(u)$ along with u, where the points are ordered by x-coordinates (hence, also by y-coordinates).

The above design completes the externalization of the structure in [7]. Next, we describe new mechanism for obtaining $O(1 + k/B)$ query time. First, we index the points in each chunk z with a few-point structure of Lemma 5. Moreover, for every proper ancestor u of z in $\mathcal{T}$, we store two sets $LMAX(z, u)$ and $RMAX(z, u)$ defined as follows. Abusing notation slightly, let $\pi(z, u)$ be the path from z to the child of u that is an ancestor of z. Also, define $\Pi_\ell(z, u)$ as the set of left siblings of the nodes on $\pi(z, u)$, and conversely, $\Pi_\gamma(z, u)$ the set of right siblings of those nodes. Then, $LMAX(z, u)$ is the skyline of the point set:

$$\bigcup_{v \in \Pi_\ell(z,u)} high(v),$$

whereas $RMAX(z, u)$ is defined symmetrically by replacing $\Pi_\ell(z, u)$ with $\Pi_\gamma(z, u)$. The points of both $LMAX(z, u)$ and $RMAX(z, u)$ are sorted by x-coordinates.

Space. Let $h = O(\log U)$ be the height of $\mathcal{T}$. We analyze first the space on the $O(U/\lambda)$ internal nodes u of $\mathcal{T}$. First, $high(u)$ fits in $O(1)$ blocks. Second, as $MAX(u)$ has $O(hB)$ points, it occupies $O(h)$ blocks. All the internal nodes thus occupy $O(h \cdot (U/\lambda)) = O(U/B) = O(n/B)$ blocks in total.

Now, let us focus on the $O(U/\lambda)$ leaf nodes z of $\mathcal{T}$. As each few-point structure uses linear space, all the few-point structures demand $O(U/\lambda + n/B) = O(n/B)$ blocks altogether. Regarding $LMAX(z, u)$, z has at most h proper ancestors u, while each $LMAX(z, u)$ requires $O(h)$ blocks. Hence, the $LMAX(z, u)$ of all z and u occupy $O((U/\lambda) \cdot h^2) = O(n/B)$ blocks in total. The case with $RMAX(z, u)$ is symmetric. The overall space consumption is therefore linear.

Query. We will need the following fact:

Lemma 6. *Given a node u in $\mathcal{T}$ and a value β, we can report the k points in the skyline of $P(u, \beta)$ in $O(1 + k/B)$ I/Os, where $P(u, \beta)$ is the set of points in $P(u)$ with y-coordinates greater than β. The points are reported in the order of y-coordinates.*

Consider a top-open query with $Q = [\alpha_1, \alpha_2] \times [\beta, U]$, where $\alpha_1, \alpha_2, \beta \in [U]$. To answer it, we first identify the trunks z_1 and z_2 that cover α_1 and α_2, respectively. This takes $O(1)$ time by dividing α_1 and α_2 by the chunk size λ, respectively. If $z_1 = z_2$, the query can be solved by searching the few-point structure of z_1 in $O(1 + k/B)$ I/Os (Lemma 5). Next, we focus on $z_1 \neq z_2$.

Let u be the lowest common ancestor of z_1 and z_2 in $\mathcal{T}$. As $\mathcal{T}$ is a *complete* binary tree, u can be determined in constant time. The rest of the algorithm proceeds in 4 steps:

Step 1. Use the few-point structure of z_2 to report the skyline of $P(z_2) \cap Q$. Let $S(z_2)$ be the set of points retrieved, and $\beta^\star$ the maximum y-coordinate of the points in $S(z_2)$. If $S(z_2) = \emptyset$, $\beta^\star = \beta$.

Step 2. Report the set S_2 of points in $LMAX(z_2, u)$ whose y-coordinates are above $\beta^\star$. Denote by $v_1, ..., v_c$ the nodes of $\Pi_\ell(z_2, u)$ in descending levels for some integer c. For each $i \in [1, c]$, check whether $high(v_i) \cap S_2$ has B points. If not, the subtree of v_i can be eliminated. Otherwise, apply Lemma 6 to retrieve the skyline of $P(v_i, \beta_i)$, where β_i is the maximum y-coordinate of the points in S_2 that are lower than $highend(v_i)$. If no such point exists, $\beta_i = \beta^\star$. If $S_2 \neq \emptyset$, update $\beta^\star$ to be the y-coordinate of the highest point in S_2.

Step 3. Find the set S_1 of points in $RMAX(z_1, u)$ whose y-coordinates exceed $\beta^\star$. Denote by $v'_1, ..., v'_{c'}$ the nodes of $\Pi_\gamma(z_1, u)$ in descending levels for some integer c'. For each $i \in [1, c']$, if $|high(v'_i) \cap S_1| = B$, apply Lemma 6 to retrieve the skyline of $P(v'_i, \beta'_i)$, where β'_i is the maximum y-coordinate of the points in S_1 that are lower than $highend(v'_i)$ (if no such point, $\beta'_i = \beta^\star$). If $S_1 \neq \emptyset$, set $\beta^\star$ to the y-coordinate of the highest point in S_1.

Step 4. Fetch the skyline of $P(z_1) \cap [\alpha_1, \alpha_2] \times [\beta^\star, U]$ from the few-point structure of z_1.

To analyze the cost, we focus on the first two steps because the other steps are symmetric. By Lemma 5, Step 1 takes $O(1 + k'/B)$ I/Os, where k' is the number of points reported in this step. In Step 2, by leveraging the ordering inside $LMAX(z_2, u)$, S_2 can be found in $O(1 + |S_2|/B)$ I/Os. We charge the second term on the points of S_2. Note that the points in S_2 are sorted by y-coordinates, thanks to the ordering in $LMAX(z_2, u)$. For each $i \in [1, c]$, if $high(v_i) \cap S_2$ has less than B points[3], the subtree of v_i incurs no more cost. Otherwise, the application of Lemma 6 takes $O(k'_i/B)$ I/Os if the application finds k'_i points (note that $k'_i \geq B$ since the whole $high(v_i)$ is definitely reported). We charge the cost on those k'_i points. Overall, every reported point is charged $O(1/B)$ I/Os. Step 1-4 each necessitate $O(1)$ extra I/Os. The total query cost is therefore $O(1 + k/B)$.

As in the proof of Lemma 6, it is easy to modify the above algorithm to output the points in the order of y-coordinates. We thus have arrived at:

Theorem 2. *There is a linear-size structure on n points in the rank space such that a top-open range skyline query can be answered optimally in $O(1 + k/B)$ I/Os, where k is the number of reported points. The points of the query result are reported in the order of y-coordinates.*

General U. The above solution is for the rank space. For general U, we can extend our solution to obtain:

Corollary 1. *There is a linear-size structure on a set of n points in $[U]^2$ (where $U \geq n$ is an integer) such that a top-open range skyline query can be answered in $O(\log \log_B U + k/B)$ I/Os, where k is the number of reported points. The points of the query result are reported in the order of y-coordinates. The query cost is optimal even without the indivisibility assumption.*

Proof. We divide the points of P into n/λ disjoint subsets by their x-coordinates, where each subset has exactly λ points, except possibly the last subset. Refer to each subset as a *chunk*. As before, build a complete binary search tree $\mathcal{T}$ on the set of trunks. All the secondary structures are exactly the same as described previously.

To answer a top-open query with $Q = [\alpha_1, \alpha_2] \times [\beta, U]$, we can also apply the same algorithm as presented earlier. The only difference is that the leaf nodes z_1 and z_2 of $\mathcal{T}$ can no longer be obtained in constant time. Instead, we find them by looking for the predecessors of α_1 and α_2 respectively, among the starting x-coordinates of all the chunks. It is well-known that a predecessor query on n values in $[U]$ can be answered in $O(\log \log_B U)$ I/Os using a structure of $O(n/B)$ blocks.

[3]This can be checked efficiently because the points of $high(v_i)$ (if any) are consecutive in S_2.

The optimality of the query time follows directly from the reduction explained in the proof of Theorem 1 and the $\Omega(\log \log_B U)$ lower bound of predecessor search under the linear space budget [30]. □

4 General Range Skyline Queries

This section will move away from top-open queries. It would be nice if there was a linear-size structure that answered a bottom-open query in $O(\log_B n + k/B)$ I/Os. Unfortunately, this is impossible. In fact, this is impossible even for anti-dominance queries, which turn out to be as hard as general (4-sided) range skyline queries when linear space is compulsory. Next, we will formally establish these facts.

A lower bound. We will first present a hardness result for anti-dominance queries. As mentioned in Section 1.2, some progress has been made in the internal-memory pointer machine. Kejlberg-Rasmussen et al. [23] proved:

Lemma 7 ([23]). *For any integer $d \geq 1$ and $\lambda \geq 1$, there is a set P of d^λ points in $\mathbb{R}^2$ and a set G of $\lambda d^{\lambda-1}$ anti-dominance queries such that (i) each query in G retrieves d points of P, and (ii) at most one point in P is returned by two queries in G simultaneously.*

We use the term (d, λ)-*input* to refer to the set P obtained in the above lemma after d and λ have been fixed. We deploy such input sets to derive:

Theorem 3. *Regarding anti-dominance queries on n points, any indivisible structure of at most cn/B blocks must incur $\Omega((n/B)^{1/(25c)} + k/B)$ I/Os answering a query in the worst case, where $c \geq 1$ is a constant and k is the result size.*

Proof. Let us first review the *indexability theorem* of [17]. Let Λ be a structure on a (d, λ)-input. Define the *access overhead* of Λ as the smallest value A that allows us to claim: Λ answers any query with output size d in Ad/B I/Os. In the context of Lemma 7, the indexability theorem states:

$$\textit{if } d \geq \tfrac{B}{2} \textit{ and } A \leq \tfrac{\sqrt{B}}{4}, \textit{ } \Lambda \textit{ must use at least } \tfrac{\lambda}{12}\tfrac{d^\lambda}{B} \textit{ blocks.}$$

Next, we will argue that if a structure has query complexity $O((n/B)^{1/(25c)} + k/B)$, it must use strictly more than cn/B blocks in the worst case. This implies that no structure of at most cn/B blocks can guarantee the aforementioned query time, and hence, proving Theorem 3.

Consider any structure with query time $O((n/B)^{1/(25c)} + k/B)$. Let Λ be the structure's instance on a (d, λ)-input where $d = B$ and $\lambda = 12c + 1.1$. The I/O cost of Λ answering a query with output size $k = d$ is at most

$$\begin{aligned} & \alpha((d^\lambda/B)^{1/(25c)} + d/B) \\ = \; & \alpha B^{\frac{12c+0.1}{25c}} + \alpha = \alpha B^{\frac{12}{25}+\frac{0.1}{25c}} + \alpha \leq \alpha B^{\frac{12.1}{25}} + \alpha \end{aligned}$$

where $\alpha > 0$ is a certain constant. It thus follows that $A \leq \alpha B^{\frac{12.1}{25}} + \alpha < \sqrt{B}/4$ when B is sufficiently large. Therefore, by the indexability theorem, the structure must occupy at least $(\lambda/12)d^\lambda/B = (c + 1.1/12)n/B > cn/B$ blocks. □

We thus have the following cleaner result:

Corollary 2. *Regarding anti-dominance queries on n points, any indivisible linear-size structure must incur $\Omega((n/B)^\epsilon + k/B)$ I/Os answering a query in the worst case, where $\epsilon > 0$ is a constant and k is the result size.*

Note that the same bound obviously holds for any generalization of anti-dominance queries, for instance, 4-sided range skyline queries.

Optimal structure. We now give an indivisible linear-size structure on a set P of n points in $\mathbb{R}^2$ such that a 4-sided range skyline query can be answered in $O((n/B)^\epsilon + k/B)$ I/Os, where $\epsilon > 0$ can be any constant. The query performance is optimal according to Corollary 2.

Store P in a B-tree T that indexes the points' x-coordinates. T has leaf capacity B and internal fanout $f = (n/B)^\epsilon / \log_B n$. The height h of T is thus $O(\log_f(n/B)) = O(1)$. For each node u in T, let $P(u)$ be the set of points in the subtree of u. We manage $P(u)$ using a structure $R(u)$ for answering right-open queries. By the symmetry of right- and top-open queries, $R(u)$ can be implemented as a structure of Theorem 1. The right-open structures of all nodes at the same level of T consume $O(n/B)$ space in total. As T has only constant levels, the total space cost is $O(n/B)$.

Given a 4-sided query with search rectangle $Q = [\alpha_1, \alpha_2] \times [\beta_1, \beta_2]$, we find in $O(h(f/B)) = O((n/B)^\epsilon)$ I/Os the leaf nodes of T containing the successor and predecessor of α_1 and α_2 respectively, among the x-coordinates indexed by T. If $z_1 = z_2$, solve the query by loading the B points in z_1 into memory using $O(1)$ I/Os.

Consider now $z_1 \neq z_2$. Let π_1 (π_2) be the path from the lowest common ancestor of z_1 and z_2 to z_1 (z_2). Let S be the set of child nodes v of the internal nodes on $\pi_1 \cup \pi_2$ such that the x-interval of v is fully contained in $[\alpha_1, \alpha_2]$ (note that every node in T corresponds to an x-interval tightly enclosing the x-coordinates in its subtree). There is a natural ordering of the nodes in S by their x-intervals. We can easily obtain S in this order using $O(h(f/B)) = O((n/B)^\epsilon)$ I/Os. Note that $|S| \leq hf = O((n/B)^\epsilon / \log_B n)$.

We will issue a 4-sided query for z_2, then a right-open query for each node in S, and finally a 4-sided query for z_1. Specifically, we first find the skyline of $P(z_2) \cap Q$. This takes constant I/Os because z_2 has only B points. Let $\beta^\star$ be the maximum y-coordinate of the points in the aforementioned skyline. Next, we process the nodes v of S in the right-to-left order of their x-intervals. For each v, perform a right-open query with $(-\infty, \infty) \times [\beta^\star, \beta_2]$ on $R(v)$, and output all the points retrieved. If the query returned at least one point, update $\beta^\star$ to be the y-coordinate of the highest point returned. Finally, issue a 4-sided query with $[\alpha_1, \alpha_2] \times [\beta^\star, \beta_2]$ on z_1 in $O(1)$ I/Os.

Since each right-open query on the nodes of S costs $O(\log_B n)$ I/Os (plus linear output time), all such queries incur in total $O(|S| \log_B n + k/B) = O((n/B)^\epsilon + k/B)$ I/Os. By the order we issued those queries and the output order of each query (guaranteed by Theorem 1), it is clear that the above algorithm can generate the result points in the order of y-coordinates in $O(k/B)$ extra I/Os. We thus conclude:

Theorem 4. *There is an indivisible linear-size structure on n points in $\mathbb{R}^2$ such that, a 4-sided range skyline query can be answered in $O((n/B)^\epsilon + k/B)$ I/Os, where k is the number of reported points. The points of the query result are reported in the order of y-coordinates. The query time is optimal under the indivisibility assumption.*

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[2] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 346–357, 1999.

[3] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems (TODS)*, 33(4), 2008.

[4] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 322–331, 1990.

[5] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 421–430, 2001.

[6] P. Bozanis, N. Kitsios, C. Makris, and A. K. Tsakalidis. New upper bounds for generalized intersection searching problems. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 464–474, 1995.

[7] G. S. Brodal and K. Tsakalidis. Dynamic planar range maxima queries. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 256–267, 2011.

[8] B. Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM (JACM)*, 37(2):200–212, 1990.

[9] B. Chazelle and D. Liu. Lower bounds for intersection searching and fractional cascading in higher dimension. *Journal of Computer and System Sciences (JCSS)*, 68(2):269–284, 2004.

[10] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 717–816, 2003.

[11] F. d'Amore, P. G. Franciosa, R. Giaccio, and M. Talamo. Maintaining maxima under boundary updates. In *Proceedings of Algorithms and Complexity, Third Italian Conference*, pages 100–109, 1997.

[12] A. S. Das, P. Gupta, A. K. Kalavagattu, J. Agarwal, K. Srinathan, and K. Kothapalli. Range aggregate maximal points in the plane. In *WALCOM*, pages 52–63, 2012.

[13] G. N. Frederickson and S. H. Rodger. A new approach to the dynamic maintaince of maximal points in a plane. *Discrete & Computational Geometry*, 5:365–374, 1990.

[14] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 1–7, 1990.

[15] R. Grossi and G. F. Italiano. Efficient splitting and merging algorithms for order decomposable problems. *Information and Computation*, 154(1):1–33, 1999.

[16] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 47–57, 1984.

[17] J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM (JACM)*, 49(1):35–55, 2002.

[18] J. Iacono and M. Patrascu. Using hashing to solve the dictionary problem. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 570–582, 2012.

[19] R. Janardan. On the dynamic maintenance of maximal points in the plane. *Information Processing Letters (IPL)*, 40(2):59–64, 1991.

[20] A. K. Kalavagattu, A. S. Das, K. Kothapalli, and K. Srinathan. On finding skyline points for range queries in plane. In *Proceedings of the Canadian Conference on Computational Geometry (CCCG)*, 2011.

[21] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 257–276, 1999.

[22] S. Kapoor. Dynamic maintenance of maxima of 2-d point sets. *SIAM Journal of Computing*, 29(6):1858–1877, 2000.

[23] C. Kejberg-Rasmussen, K. Tsakalidis, and K. Tsichlas. I/o-efficient dynamic planar range skyline queries. *Manuscript*, 2012.

[24] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of Very Large Data Bases (VLDB)*, pages 275–286, 2002.

[25] K. G. Larsen and R. Pagh. I/O-efficient data structures for colored range and prefix reporting. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 583–592, 2012.

[26] M. D. Morse, J. M. Patel, and H. V. Jagadish. Efficient skyline computation over low-cardinality domains. In *Proceedings of Very Large Data Bases (VLDB)*, pages 267–278, 2007.

[27] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences (JCSS)*, 23(2):166–204, 1981.

[28] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 214–221, 1993.

[29] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 467–478, 2003.

[30] M. Patrascu and M. Thorup. Time-space trade-offs for predecessor search. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[31] A. D. Sarma, A. Lall, D. Nanongkai, and J. Xu. Randomized multi-pass streaming skyline algorithms. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):85–96, 2009.

[32] M. Sharifzadeh, C. Shahabi, and L. Kazemi. Processing spatial skyline queries in both vector spaces and spatial network databases. *ACM Transactions on Database Systems (TODS)*, 34(3), 2009.

[33] C. Sheng and Y. Tao. Finding skylines in external memory. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 107–116, 2011.

[34] S. Subramanian and S. Ramaswamy. The p-range tree: A new data structure for range searching in secondary memory. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 378–387, 1995.

[35] Y. Tao. Indexability of 2d range search revisited: constant redundancy and weak indivisibility. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 131–142, 2012.

[36] Y. Tao. Stabbing horizontal segments with rays. In *Symposium on Computational Geometry (SoCG)*, 2012.

[37] E. Verbin and Q. Zhang. The limits of buffering: a tight lower bound for dynamic membership in the external memory model. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 447–456, 2010.

[38] J. S. Vitter. Algorithms and data structures for external memory. *Foundation and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[39] H. Yuan and M. J. Atallah. Data structures for range minimum queries in multidimensional arrays. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 689–700, 2010.

Appendix 1: Proof of Lemma 3

Our argument is by induction on the position of ℓ. For this purpose, care must be taken to interpret the rectangles of the nodes currently in $T(\ell)$. As these nodes have not been version copied yet, the right edges of their rectangles lie on ℓ. As ℓ moves, so do these right edges, until the corresponding nodes are version copied. Let set $\Sigma_1(\ell)$ include the bottom edges of the rectangles of all leaf nodes already spawned so far, counting also the nodes in $T(\ell)$. Remember that the right endpoints of those edges are not taken, as explained earlier. When we finish building all the leaves, $\Sigma_1(\ell)$ becomes the final Σ_1. We will show that $\Sigma_1(\ell)$ is nesting and monotonic at all times. This is obviously true when ℓ is at $x = -\infty$.

Now, suppose that $\Sigma_1(\ell)$ is currently nesting and monotonic. We will prove that it remains so after the next update on $T(\ell)$. This is trivial if the update does not cause any version copy, i.e., the first leaf node u of $T(\ell)$ is not full yet. Consider instead that u is version copied to u'. At this point, $r(u)$ is finalized. Because $r(u)$ is the lowest among the rectangles of the nodes in $T(\ell)$, its finalization cannot affect the nesting and monotonicity of $\Sigma_1(\ell)$. The version copy also creates $r(u')$. Note that the right edge of $r(u)$ and the left edge of $r(u')$ both lie on ℓ. Hence, the bottom edge of $r(u)$ is disjoint with that of $r(u')$ (recall that the right endpoint of an edge is not taken). Furthermore, $r(u')$ has the same y-interval as $r(u)$, and a zero-length x-interval (i.e., the x-interval is a point). Therefore, if no split/merge follows, $\Sigma_1(\ell)$ is still nesting and monotonic.

Next, consider that u' is split into u'_1 and u'_2. In this case, $r(u')$ disappears from $\Sigma_1(\ell)$, and is replaced by $r(u'_1)$ and $r(u'_2)$, which are the bottom two among the rectangles of the nodes in $T(\ell)$. Furthermore, both $r(u'_1)$ and $r(u'_2)$ have zero-length x-intervals that degenerate into a point at the position of ℓ. It follows that $\Sigma_1(\ell)$ is still nesting and monotonic.

It remains to discuss the case where u' needs to be merged with its sibling v. When this happens, the algorithm first version copies v to v', which finalizes $r(v)$. The x-interval of $r(v)$ must contain that of $r(u)$, which is consistent with nesting and monotonicity because $r(v)$ is above $r(u)$. The merge of u' and v' creates a node z, such that $r(z)$ has zero-length x-interval. Note that $r(z)$ is

currently the lowest of the rectangles of the nodes in $T(\ell)$. It is clear that $\Sigma_1(\ell)$ remains nesting and monotonic.

Finally, z may still need to be split one more time, but this case can be analyzed in the same way as the split scenario mentioned earlier. We thus conclude the proof.

Appendix 2: Proof of Lemma 6

If u is a leaf, find the skyline of $P(u, \beta)$ by issuing a top-open query with search rectangle $[-U, U] \times [\beta, U]$ on the few-point structure of u. The query time is $O(1 + k/B)$ by Lemma 5.

The rest of the proof is the adaptation of a similar argument in [7] to external memory. If u is internal, we find the skyline of $P(u, \beta)$ as follows. Load $high(u)$ into memory, and report the points therein with y-coordinates above β. If there are less than B such points, we have found the entire skyline of $P(u, \beta)$, due to the definition of $high(u)$.

Suppose instead that the entire $high(u)$ is reported. Let $p = highend(u)$. It suffices to consider the points that

- (i) are in the subtrees of the nodes in $\Pi_\gamma(u)$, or
- (ii) share the same trunk as, but are to the right of, p.

Any other point of $P(u)$ must be either in $high(u)$ (which is already visited) or dominated by p.

To find the skyline points in (i), we collect the set S of points in $MAX(u)$ whose y-coordinates are above β. The points in S need to be returned, but there can be more result points in (i). To extract them all, we need to explore the subtrees of certain nodes in $\Pi_\gamma(u)$. Specifically, let $v_1, ..., v_c$ be the nodes in $\Pi_\gamma(u)$ in ascending levels, where c is some integer. For each v_i, if $S_i = high(v_i) \cap S$ has less than B points, the subtree of v_i can be pruned from further consideration. Otherwise (i.e., the whole $high(v_i)$ is in S), we recursively report the skyline of $P(v_i, \beta_i)$, where β_i is the maximum y-coordinate of points in S that are lower than $highend(v_i)$. If no such point exists, $\beta_i = \beta$.

The skyline points in (ii) can be retrieved with a top-open query on the few-point structure of the trunk z covering x_p, where z can be reached in constant time following a pointer stored at u. Specifically, compute β_0 to be the maximum y-coordinate of the points in S (if $S \neq \emptyset$), or β (if $S = \emptyset$). The top-open query for z has rectangle $(x_p, max\text{-}x(z)] \times (\beta_0, U]$, where $max\text{-}x(z)$ is the maximum x-coordinate covered by z.

Now we analyze the query cost. If less than B points of $high(u)$ are reported, the algorithm finishes with 1 I/O. Otherwise, the scan of $MAX(u)$ takes $O(1+|S|/B)$ I/Os. If $|S| < B$, we charge the $O(1 + |S|/B) = O(1)$ cost on the B points in $high(u)$; otherwise, we charge the $O(|S|/B)$ cost on the points of S. The top-open query on the few-point structure of z requires $O(1 + k'/B)$ I/Os if it returns k' points. If $k' < B$, we charge the $O(1 + k'/B) = O(1)$ cost on the points of $high(u)$; otherwise, charge the $O(k'/B)$ I/Os on the k' points.

It remains to discuss the I/Os spent on $v_1, ..., v_c$. For each $i \in [1, c]$, if $|S_i| < B$, there is no cost on v_i. Otherwise, we charge to the points of S_i the $O(1)$ I/Os needed to read (i) $high(v_i)$ and (ii) the first block of $MAX(v_i)$. Overall, every reported point is charged $O(1/B)$ I/Os. The total query time is therefore $O(1 + k/B)$.

A final remark concerns the output ordering, which has been ignored by the above algorithm in order to keep the presentation simple. However, it is easy to modify the algorithm to ensure the desired ordering. First, the points of $high(u)$ are clearly the highest in the result, and hence, are reported first. Consider the moment when we have obtained $S_1, ..., S_c$ (recall that their union is

S). For each $i \in [2, c]$, we do not report S_i until all the points from the subtree of v_{i-1} have been output. This means that we report S_i only after (i) if $|S_{i-1}| < B$, S_{i-1} has been output, or (ii) otherwise, the skyline of $P(v_{i-1}, \beta_i)$ has been output.